# EE 457 Unit 8

## Exceptions

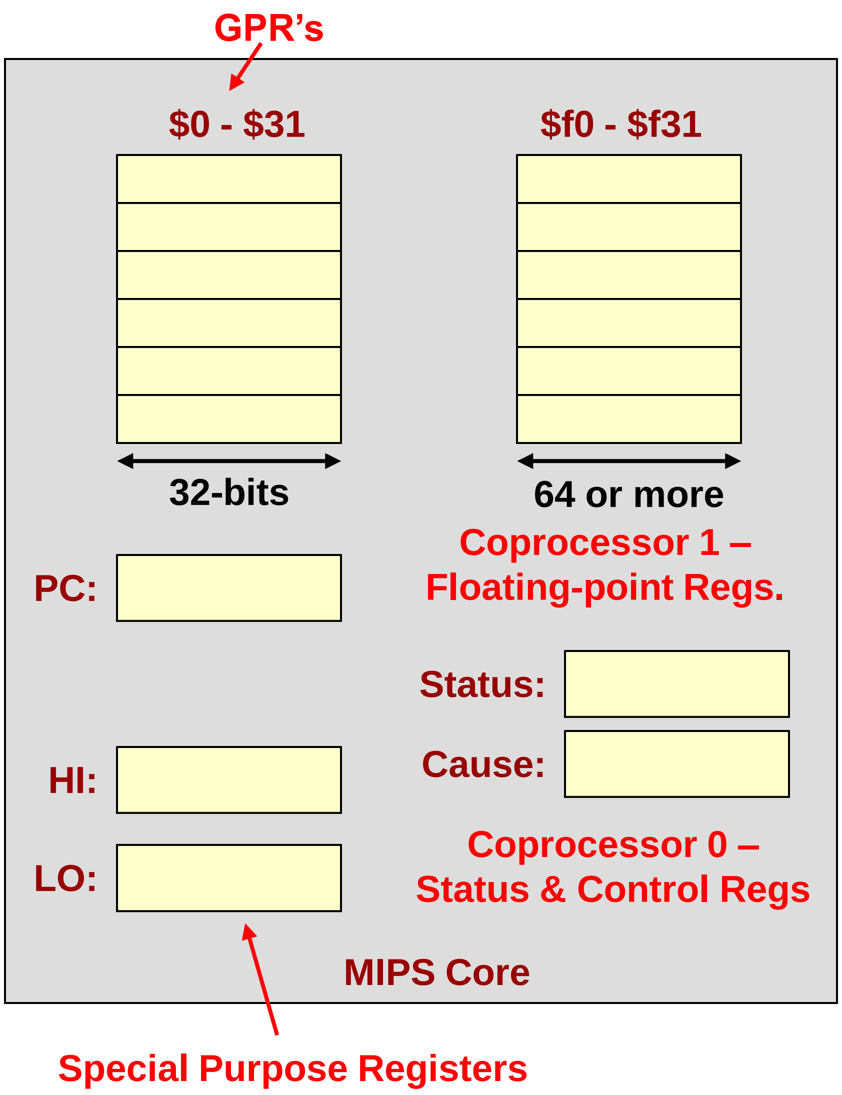
"What Happens When Things Go Wrong"

User and Kernel Mode, Coprocessor Registers

# BACKGROUND

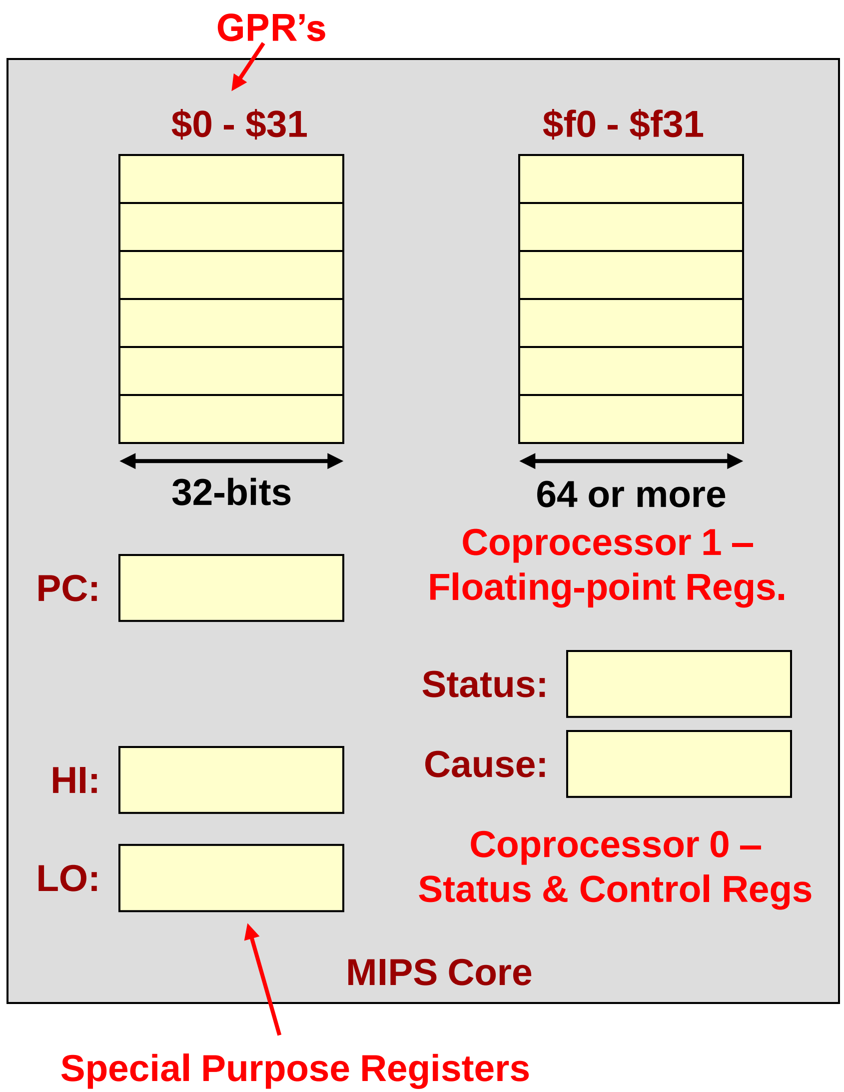
# MIPS Programmer-Visible Registers

- Normal MIPS instructions CANNOT access coprocessor registers directly (since instruction format does not have enough bits)
- Coprocessor registers can be accessed via the `mfc0` (move from c0) and `mtc0` (move to c0) instructions
- `mfc0    $gpr,$c0_reg`
  - R[gpr]  = C0[c0_reg]
- `mtc0    $gpr,$c0_reg`
  - C0[c0_reg] = R[gpr]
- Sequence:
  - Move value from coprocessor register to normal GPR
  - Process that value with regular MIPS instructions
  - Move value back to coprocessor register



# MIPS Coprocessor Register Access

- Normal MIPS instructions CANNOT access coprocessor registers directly (since instruction format does not have enough bits)
- Coprocessor registers can be accessed via the `mfc0` and `mtc0` instructions
- `mfc0 $gpr,$c0_reg`
  - R[gpr] = C0[c0_reg]
- `mtc0 $gpr,$c0_reg`
  - C0[c0_reg] = R[gpr]
- Sequence:
  - Move value from coprocessor register to normal GPR
  - Process that value with regular MIPS instructions
  - Move value back to coprocessor register

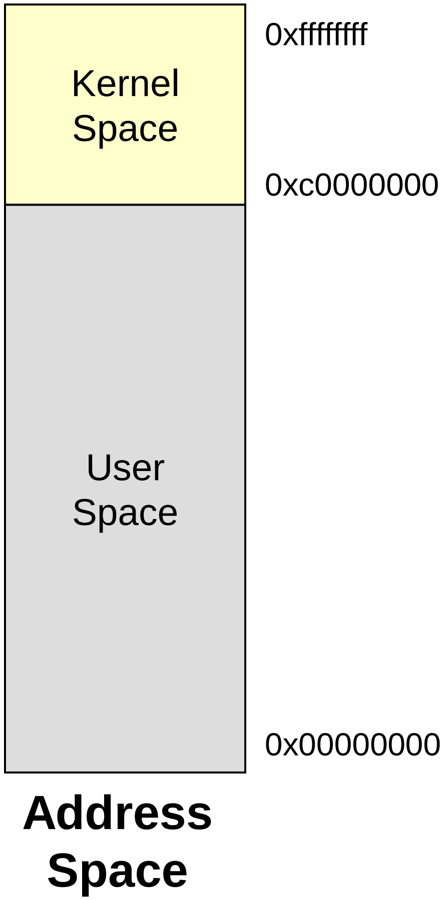


# User vs. Kernel Mode

- Kernel mode is a special mode of the processor for executing trusted (OS) code
  - Certain features/privileges are only allowed to code running in kernel mode
  - OS and other system software should run in kernel mode
- User mode is where user applications are designed to run to limit what they can do on their own
  - Provides protection by forcing them to use the OS for many services
- User vs. kernel mode determined by some bit(s) in some processor control register
  - x86 Architecture uses lower 2-bits in the CS segment register (referred to as the Current Privilege Level bits [CPL])
  - MIPS: User Mode bit in the processor status register
- **On an exception, the processor will automatically switch to kernel mode**

# Kernel Mode Privileges

- Privileged instructions
  - User apps. shouldn't be allowed to disable/enable interrupts, change memory mappings, etc.
- Privileged Memory or I/O access
  - Processor supports special areas of memory or I/O space that can only be accessed from kernel mode
- Separate stacks and register sets
  - MIPS processors can use "shadow" register sets (alternate GPR's when in kernel mode).



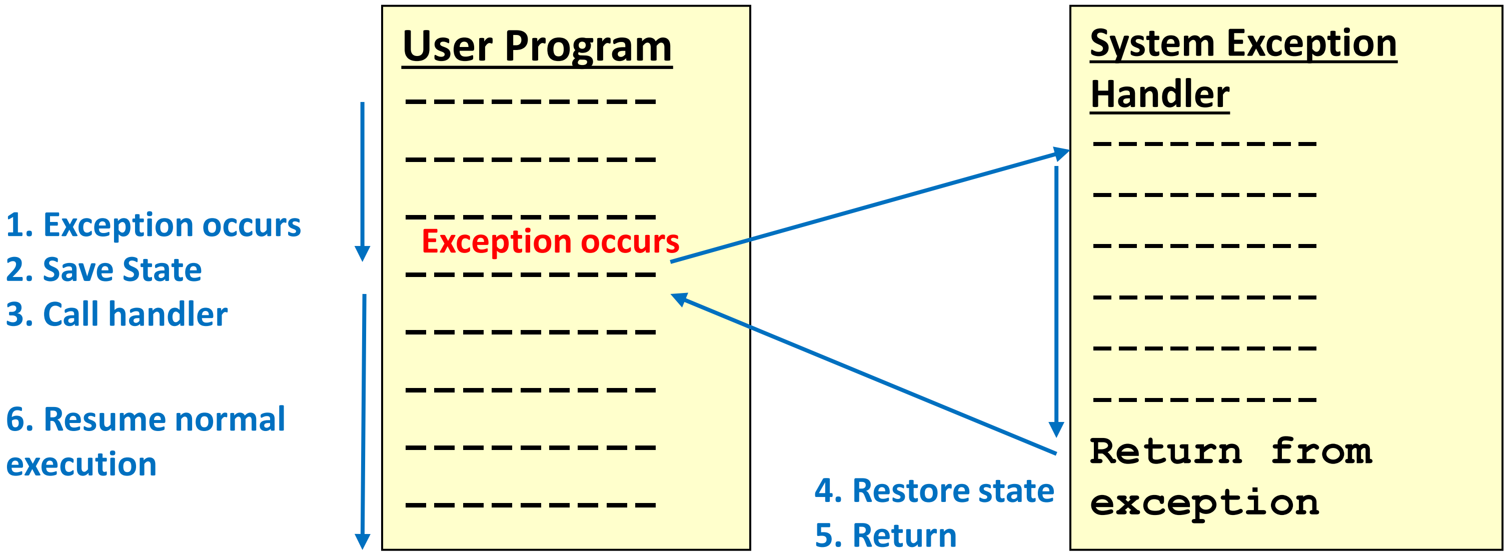
# EXCEPTIONS OVERVIEW

# What are Exceptions?

- Exceptions are "rare" events that cause a break in normal program execution
- Can be **synchronous** (called by an instruction in the program)
  - Traps or system calls (calls from user apps to OS functions)
- Can be **asynchronous** (triggered by the hardware)
  - HW Interrupts
  - Error conditions
- Response: The processor hardware must call a "predetermined" **software** routine (aka "handler" or "service routine")

# Exception Processing

- Exception handling is similar to a subroutine ('jal') call but performed automatically by the hardware
  - Must **save PC** of offending instruction, program state, and any information needed to return afterwards
  - Flush the pipeline using the hardware already present for branches/jumps
  - Execute the software handler by loading the PC with its start address (must be preset or looked up by the hardware without help from software)
  - Execute the handler routine to deal with the exception
  - Return and restore the state



# Sync. Exceptions: System Calls/Traps

- A controlled-method for user application calling OS services
- Switches processor to "kernel" mode of the processor where certain privileges are enabled that we would not want normal user apps to access

**MIPS System call**

```
addi $v0,$0,5 // $v0 = service num.
syscall       // enter OS
```

**x86 System Call (old DOS OS call)**

```
IN  AH, 01H
INT 20H     // getchar()
```

# Exception Examples 1

| Example | Stage | Action |
|---|---|---|
| **I/O Device Interrupt**<br>• A peripheral device requires action from the CPU (Interrupt I/O Driven) | WB | Take ASAP |
| **Operating System Calls ("Traps") [e.g. File Open]**<br>• Trap instruction causes processor to enter kernel mode | ID | Precise |
| **Instruction Tracing and Breakpoints**<br>• When TRAP Bit is set all instructions cause exceptions<br>• Particular instructions are flagged for exceptions (debugging) | ID | Precise |
| **Arithmetic Exceptions**<br>• Overflow or Divide-by-0 | EX | Precise |

# Exception Examples 2

| Example | Stage | Action |
|---|---|---|
| **Page Faults**<br>• Virtual memory access fault (no Page Table entry resident in memory) | IF or MEM | Precise |
| **Misaligned Memory Address**<br>• Address is not multiple of operand size | EX | Abort Process |
| **Memory Protection Violations**<br>• Address is out of bounds; RWX violation | IF or MEM | Abort Process |
| **Undefined Instructions**<br>• Decode unit does not recognize opcode or other fields<br>• Could be useful to extend the instruction set | ID | Precise (Why not abort) |
| **Hardware failure**<br>• Unrecoverable hardware error is detected; execution is compromised | WB | Take ASAP |
| **Power Failure**<br>• Power has fallen below a threshold; Trap to software to save as much state as possible | WB | Take ASAP |

# Review: Exception Processing

- Save necessary state to be able to restart the process
  - Save PC of offending instruction
- Call an appropriate "handler" routine to deal with the error / interrupt / syscall
  - Handler identifies cause of exception and handles it
  - May need to save more state
- Restore state and return to offending application (or kill it if recovery is impossible)

# MIPS Coprocessor 0 Registers

- Status Register
  - Enables and disables the handling of exceptions/interrupts
  - Controls user/kernel processor modes
    - Kernel mode allows access to certain regions of the address space and execution of certain instructions
- Cause Register: Indicates which exception/interrupt occurred
- Exception PC (EPC) Register
  - Indicates the address of the instruction causing the exception
  - This is also the instruction we should return to after handling the exception (similar to $ra ($31) for jal )

# Status Register

- Allows software to understand the state of the processor and to control whether certain exceptions (interrupts) are ignored
- Register 12 in coprocessor 0
  - IM[7:0] – Interrupt Mask bits (1 = ignore / 0 = allow)
  - UM – User Mode (1 = User mode / 0 = Kernel Mode)
  - ERL/EXL = Exception/Error Level
    - 1 = Already handling exception or error / 0 = Normal exec.
    - If either bit is '1' processor is also said to be in kernel mode
  - IE = Interrupt Enable
    - 1 = Allow unmasked interrupts / 0 = Ignore all interrupts

| | 31 | 15 | | | | | | | 8 | | 4 | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Status Register | 0000 0000 0000 0000 | IM7 | IM6 | IM5 | IM4 | IM3 | IM2 | IM1 | IM0 | 0 0 0 | UM | 0 0 | ERL | EXL | IE |

# EPC Register

- Exception PC holds the address of the offending instruction
  - Can be used along with 'Cause' register to find and correct some error conditions
- **'eret'** instruction used to return from exception handler and back to execution point in original code (unless handling the error means having the OS kill the process)
  - 'eret' Operation: PC = EPC

EPC = Exception PC

31 0

**Address of instruction that generated the exception**

# Cause Register

- Register 13 in coprocessor 0
- Bit definitions
  - BD – Branch Delay
    - The offending instruction was in the branch delay slot
    - EPC points at the branch but it was EPC+4 that caused the exception
  - PI[7:0] – Pending Interrupt
    - 1 = Interrupt Requested / 0 = No interrupt requested
  - Exception Code – Indicates cause of exception (see table)

| Code | Cause |
|---|---|
| 0 | Interrupt (HW) |
| 4, 5 | Load (4), Store (5) Address Error |
| 6, 7 | Instruc. (6), Data (7) Bus Error |
| 8 | Syscall |
| 9 | Breakpoint |
| 10 | Reserved Instruc. |
| 11 | CoProc. Unusable |
| 12 | Arith. Overflow |
| 13 | Trap |
| 15 | Floating Point |

Cause Register

| 31 | | 15 | | | | | | | 8 | 7 | 6 … 2 | 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BD | 000 0000 0000 0000 | PI7 | PI6 | PI5 | PI4 | PI3 | PI2 | PI1 | PI0 | 0 | Exception Code | 0 0 |

# Problem of Calling a Handler

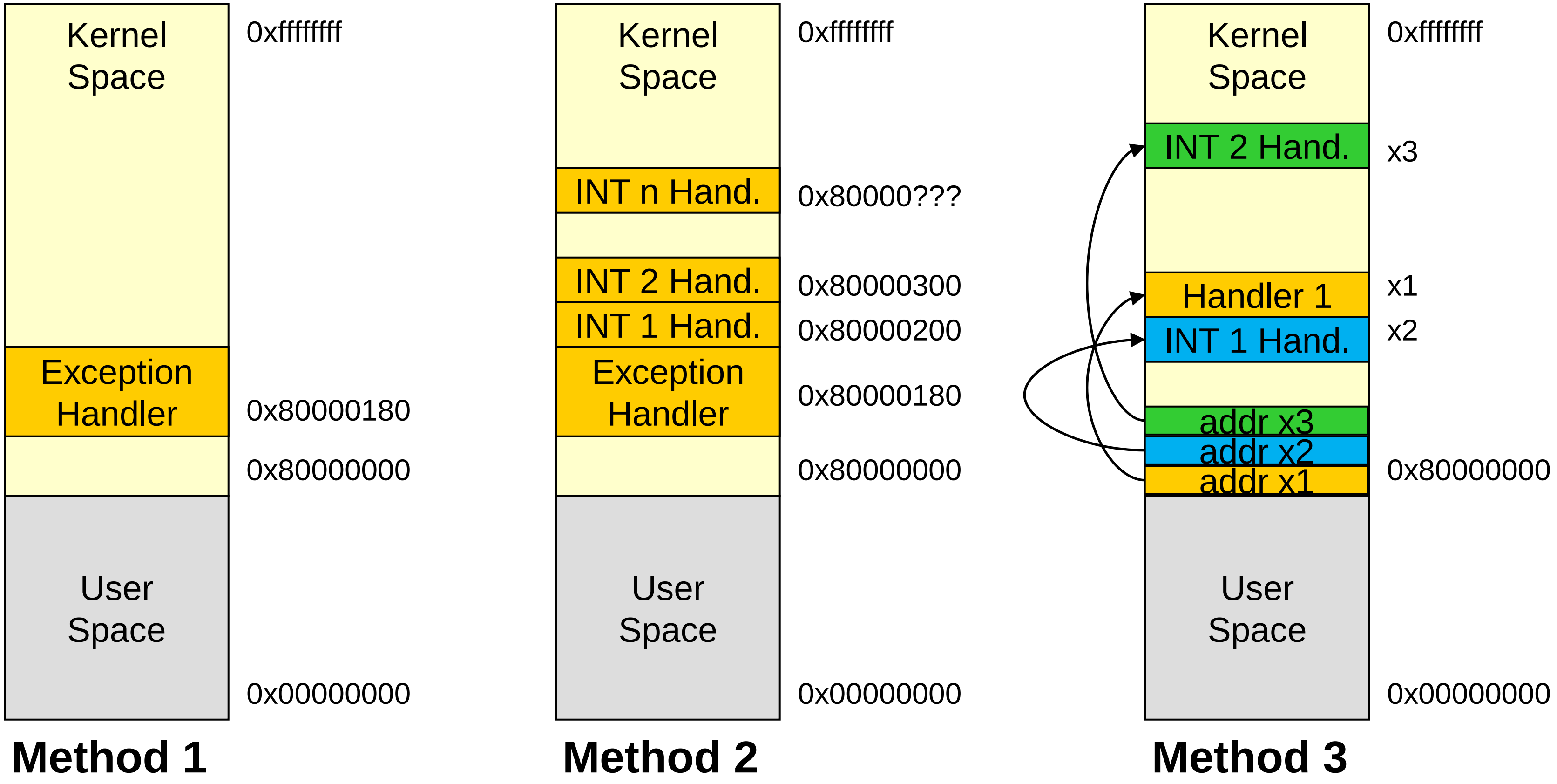
- We can't use explicit `jal` instructions to call exception handlers since we don't when they will occur

```
         .text
MAIN:    ----
         ----
         ----
         ----
         ----
         jr      $ra
```

**Many instructions could cause an error condition. Or a hardware event like a keyboard press could occur at any point in the code.**

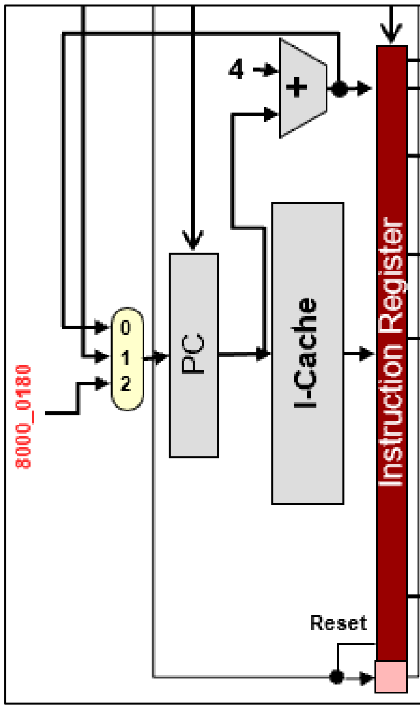
# Handler Calling Methods

- Since we don't know when an exception will occur there must be a preset location where an exception handler should be defined or some way of telling the processor in advance where our exception handlers will be located



# Solution for Calling a Handler

- Method 1: Single hardwired address for master handler
  - Early MIPS architecture defines that the exception handler should be located at 0x8000_0180. Code there should then examine CAUSE register and then call appropriate handler routine
- Method 2: Vectored locations (usually for interrupts)
  - Each interrupt handler at a different address based on interrupt number (a.k.a. vector) (INT1 @ 0x80000200, INT2 @ 0x80000300)
- Method 3: Vector tables
  - Table in memory holding start address of exception handlers (i.e. overflow exception handler pointer at 0x0004, FP exception handler pointer at 0x0008, etc.)



# "PRECISE" EXCEPTIONS

# Why are Exceptions So Important?

- Exceptions are part of the ISA (Instruction Set Architecture) specification
- Any implementation of an ISA must comply with its "Exception model"
- Precise exception handling constrains what the architecture can do
  - Exceptions are rare yet we must functionally support them
  - If we did not have to comply to the exception model, architects would have a lot more freedom in their design

**When designing micro-architectures for the common case, exceptions must always be in the back of your mind!**

# Precise Exceptions

- Precise Exceptions: A pipelined or advanced out-of-order execution processor's exception handling should **behave equivalently** to exceptions on a single-cycle CPU.
  - Any instructions BEFORE the offending instruction should complete before the handler runs
  - Any instructions AFTER the offending instruction should not appear to have executed (written to memory or register)
- Very difficult in architectures in which multiple instruction execute concurrently (i.e. our 5-stage pipeline)

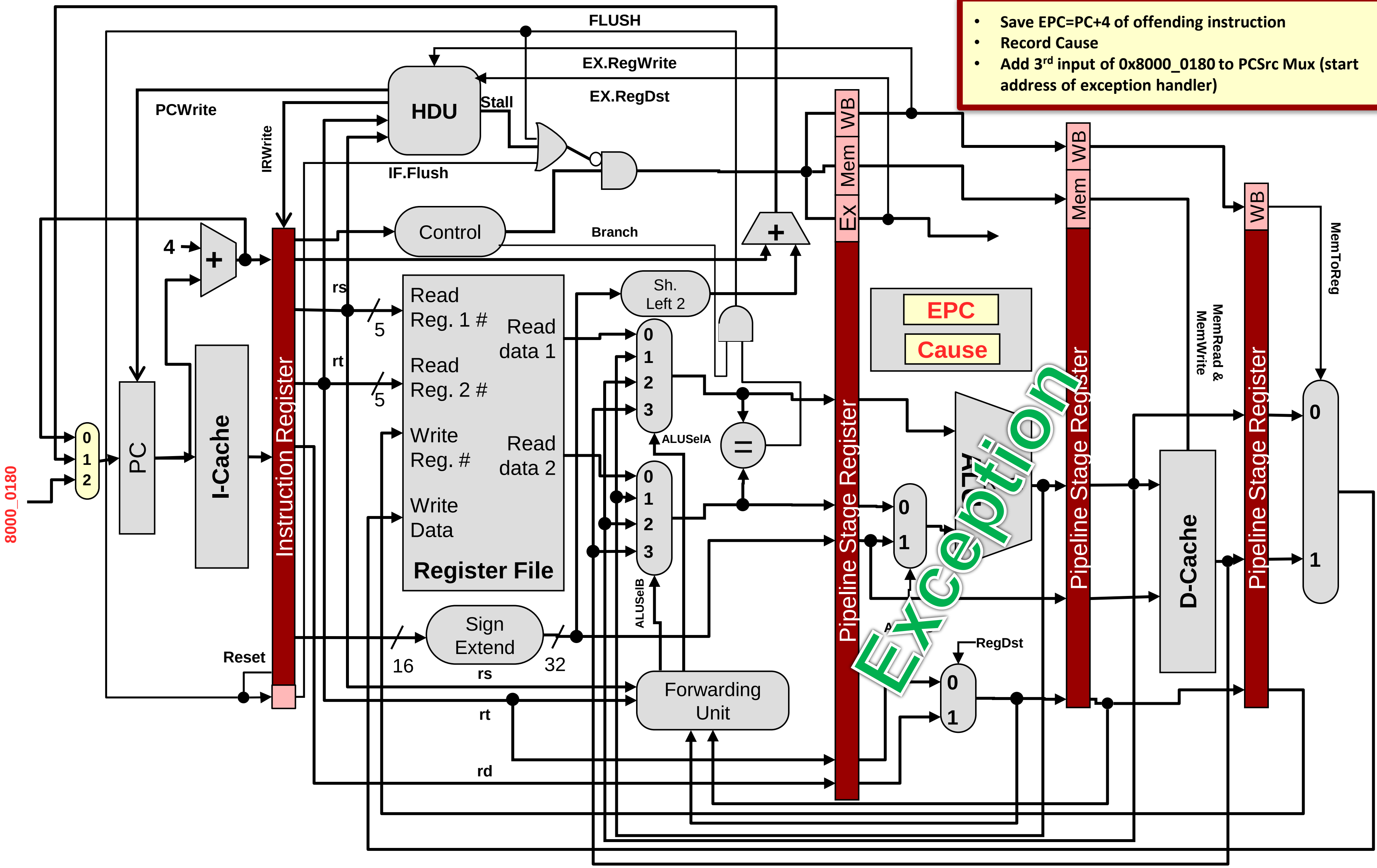
# Exceptions in the 5-Stage Pipeline

- To support precise exceptions in the 5-stage pipeline we must…
  - Identify the pipeline stage and instruction causing the exceptions
    - Any stage can trigger an exception (except for the WB stage)
  - Identify the cause of the exception
  - Save the process state at the faulting instruction
    - Including registers, PC, and cause
    - Usually done by software exception handler
  - Complete the execution of instructions preceding the faulting instruction
  - Flush instruction following the faulting instruction plus the faulting instruction
  - Transfer control to exception handler

**Use many of the same mechanisms as conditional branches.**

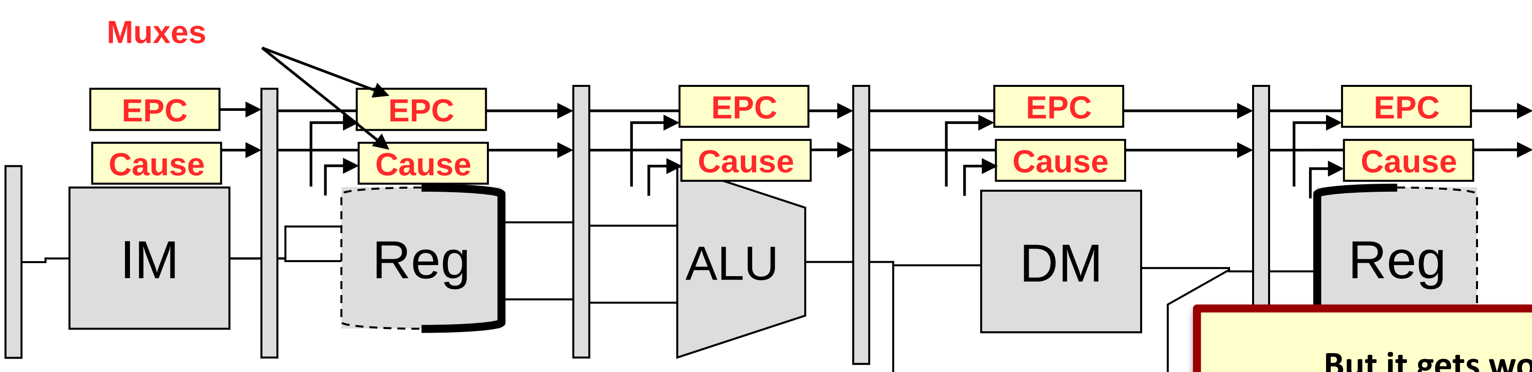
# Exception in EX stage

- Save EPC=PC+4 of offending instruction
- Record Cause
- Add 3rd input of 0x8000_0180 to PCSrc Mux (start address of exception handler)



# Exception Handling Complexities

- When the arithmetic exception is triggered in EX, we must flush IF, ID and EX and start fetching from 0x8000_0180
- Note that the handler's software must have access to CAUSE and EPC registers to figure out what to do
- Realize though exceptions may occur in all but the WB stage
  - 4 possible values of CAUSE and EPC
  - Software needs to know which value is the actual cause and EPC
  - Depending on the stage where the exception occurs, we have to flush different stages




**But it gets worse!!!**

# More Complex Complexities?

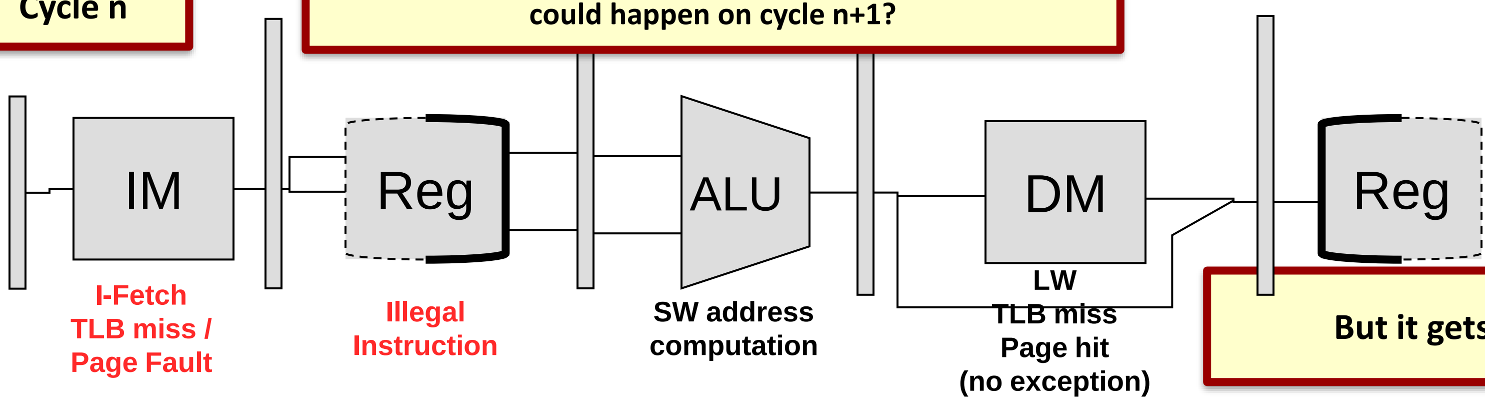
- What happens if multiple exceptions occur in the same cycle from different instructions in different stages
  - Should take the "oldest" exception in "program/process order"
  - "Program/process order" = Order if only 1 instruction were executed at a time (= Fetch order)
  - Thus oldest instruction is the one deepest (furthest) into the pipeline
  - There is no point in dealing with all exceptions, just the oldest one
  - Let software deal with the oldest and then restart…if later instruction were going to generate an exception, then they will again upon restart and we can handle it then

**Program Order**

```
xor $9, $9, $9
lw $2, 0($3)
sw $4, 0($3)
illegal
add $5, $6, $7
```

**Cycle n**

**We can start handling illegal instruction exception. What could happen on cycle n+1?**



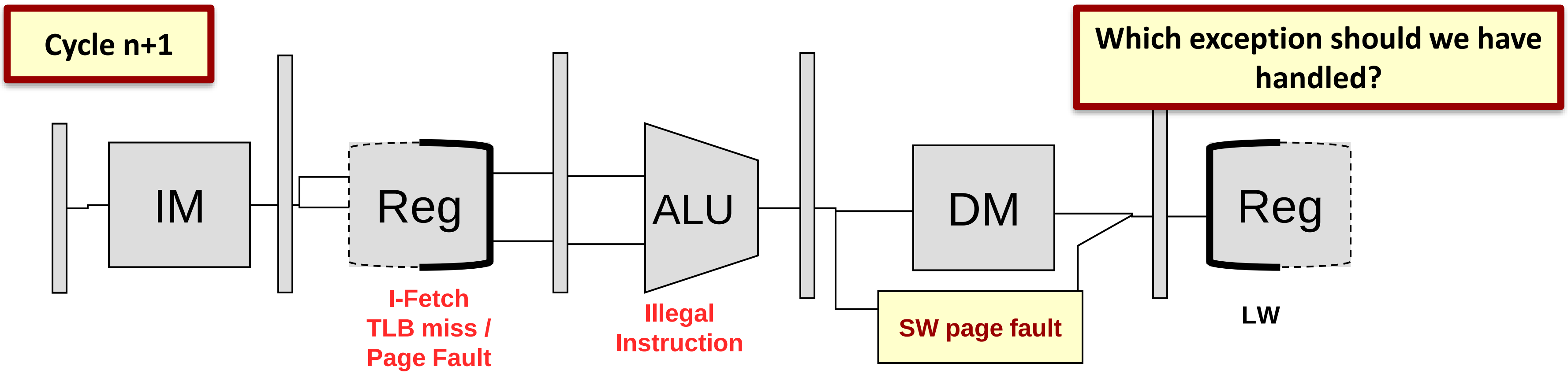

**But it gets worse!!!**

# More Complex Complexities?

- Remember we must complete instruction preceding the faulting instruction
- Remember we are supposed to handle exceptions in program order (not temporal order)

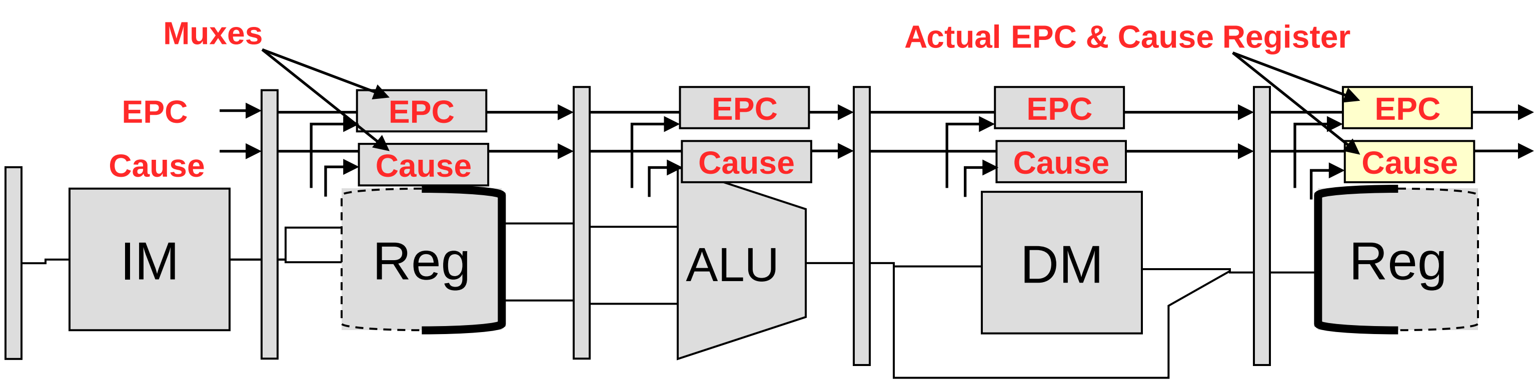
Program Order | Temporal Order

```
xor $9, $9, $9
lw $2, 0($3)
sw $4, 0($3)      2
illegal           1a
add $5, $6, $7    1b
```



# Simplify the Process

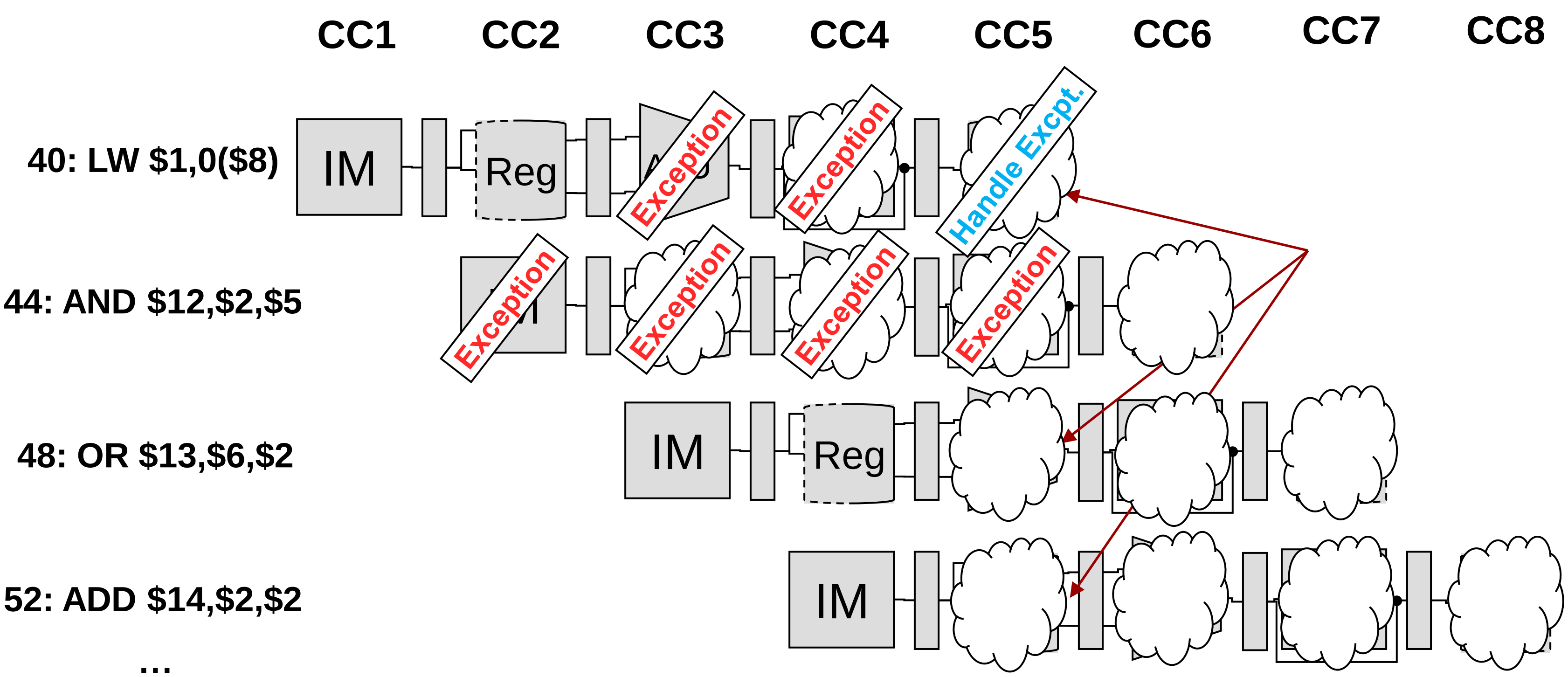
- It is not practical to take an exception in the cycle when it happens
  - Multiple exceptions in the same cycle
  - It is complex to take exception in various pipeline stages since we have to take them in program order and not temporal order
- Instead, we will just tag an instruction in the pipeline if it causes and exception (recording the cause and EPC)
  - Turn the offending instruction into a NOOP (bubble)
  - Let the instructions continue to flow down the pipeline and handle the offending instruction's execution in the WB stage
    - The cause and status info is carried down the pipe via stage registers
  - Exception remains "silent" until it reaches the WB stage
  - Exceptions are then processed into the WB stage



USC Viterbi School of Engineering

# Handling in WB Stage

- Handling in WB stage helps deal with temporal vs. program order issues



# Simplified Processing

- Precise exceptions are now taken in WB along with other HW interrupts
- Faulting instructions "carry" their cause and EPC values through the pipeline stage registers
- Only one set of EPC and CAUSE registers in the WB stage
- When an instruction flagged as faulting reaches the WB stage
  - Flush IF, ID, EX, MEM
    - Make sure that if a SW is in MEM stage that it is not allowed to write
  - Load the handler address in the PC
  - Make sure EPC & Cause are software-readable (movable to GPR's)

**This is a general approach to dealing with exceptions in the processor:**
**Wait until the faulting instruction exits the machine to trigger the handling procedure**